Zionomicon

John De Goes and Adam Fraser

Chapter 1

Foreword by John A. De Goes

We live in a complex and demanding cloud-native. The applications that we develop are small parts of a much larger, globally distributed whole.

Modern applications must process a never-ending stream of new data and requests from all around the world, interacting with hundreds or even thousands of other services, remotely distributed across servers, racks, data centers and even cloud providers.

Modern applications must run 24x7, deal with transient failures from distributed services, and respond with ultra low latency as they process the data and requests from desktops, smartphones, watches, tablets, laptops, devices, sensors, APIs, and external services.

The complexity of satisfying these requirements gave birth to *reactive programming*, which is a style of designing applications so they are responsive, resilient, elastic, and event-driven.

This is the world that created *ZIO*, a new library that brings the power of functional programming to deliver a powerful new approach to building modern applications.

1.1 A Brief History of ZIO

On June 5th, 2017, I opened an issue in the *Scalaz* repository on Github, arguing that the next major version of this library needed a powerful and fast data type for asynchronous programming. Encouraged to contribute to Scalaz by my friend Vincent Marquez, I volunteered to build one.

This was not my first foray into the space of asynchronous computing. Previously, I had designed and built the first version of *Aff*, and assisted the talented Nathan Faubion with the second iteration.

The *Aff* library quickly became the number one solution for async and concurrent programming in the Purescript ecosystem, offering powerful features not available in Javascript Promises.

Prior to *Aff*, I had written a Future data type for Scala, which I abandoned after the Scala standard library incorporated a much better version. Before that, I wrote a Promise data type for the haXe programming language, and a Raincheck data type for Java.

In every case, I made mistakes, and subsequently learned... to make new and different mistakes!

That sunny afternoon in June 2017, I imagined spending a few months developing this new data type, at which point I would wrap it up in a tidy bow, and hand it over to the Scalaz organization.

What happened next, however, I never could have imagined.

1.2 The Birth of ZIO

As I spent free time working on the core of the new project, I increasingly felt like the goal of my work should not be to just create a pure functional effect wrapper for asynchronous side-effects.

That had been done before, and while performance could be improved over Scalaz 7, libraries built on pure functional programming cannot generally be as fast as bare-metal, abstraction-free, hand-optimized procedural code.

Now, for developers like me who are sold on functional programming, an effect-wrapper is enough, but if all it does it slap a *Certified Pure* label on imperative code, it's not going to encourage broader adoption. Although solving the async problem is still useful in a pre-Loom world, lots of other other data types already solved this problem, such as Scala's own Future.

Instead, I thought I needed to focus the library on concrete pains that are well-solved by functional programming, and one pain stood out among all others: concurrency including the unsolved problem of how to safely cancel executing code whose result is no longer needed, due to timeout or other failures.

Concurrency is a big space. Whole libraries and ecosystems have been formed to tame its wily ways. Indeed, some frameworks become popular precisely because they shield developers from concurrency, because it's complex, confusing, and error-prone.

Given the challenges, I thought I should directly support concurrency in this new data type, and give it features that would be impossible to replicate in a

procedural program without special language features.

This vision of a powerful library for safe concurrent programming drove my early development.

1.3 Contentious Concurrency

At the time I began working on the Scalaz 8 project, the prevailing dogma in the functional Scala ecosystem was that an effect type should have little or no support for concurrency.

Indeed, some argued that effect concurrency was inherently unsafe and must be left to streaming libraries, like FS2 (a popular library for doing concurrent streaming in Scala).

Nonetheless, having seen the amazing work coming out of Haskell and F#, I believed it was not only possible but very important for a modern effect type to solve four closely related concurrency concerns:

- Spawning a new independent 'thread' of computation
- Asynchronously waiting for a 'thread' to finish computing its return value
- Automatically canceling a running 'thread' when its return value is no longer needed
- Ensuring cancellation does not leak resources

In the course of time, I developed a small prototype of what became known as the *Scalaz 8 IO monad*, which solved these problems in a fast and purely functional package.

In this prototype, effects could be *forked* to yield a *fiber* (a cooperatively-yielding virtual thread), which could be joined or instantly interrupted, with a Haskell-inspired version of *try/finally* called *bracket* that provided resource safety, even in the presence of asynchronous or concurrent interactions.

I was very excited about this design, and I talked about it publicly before I released the code, resulting in some backlash from competitors who doubted resource safety or performance. But on November 16, 2017, I presented the first version at Scale by the Bay, opening a pull request with full source code, including rigorous tests and benchmarks, which allayed all concerns.

Despite initial skepticism and criticism, in time, all effect systems in Scala adopted this same model, including the ability to launch an effect to yield a fiber, which could be safely interrupted or joined, with support for finalizers to ensure resource safety.

This early prototype was not yet *ZIO* as we know it today, but the seeds of ZIO had been planted, and they grew quickly.

1.4 Typed Errors & Other Evolution

My initial design for the Scalaz 8 IO data type was inspired by the Haskell IO type and the Task type from Scalaz 7. In due time, however, I found myself reevaluating some decisions made by these data types.

For one, I was unhappy with the fact that most effect types have dynamically typed errors. The compiler can't help you reason about error behavior if you pretend that every computation can always fail in infinite ways.

As a statically-typed functional programmer, I want to use the compiler to help me write better code. I can do a better job if I know where I have and have not handled errors, and if I can use typed data models for business errors.

Of course, Haskell can sort of give you statically-typed errors with monad transformers, and maybe type classes. Unfortunately, this solution increases barriers to entry, reduces performance, and bolts on a second, often confusing error channel.

Since I was starting from a clean slate in a different programming language, I had a different idea: what if instead of rigidly fixing the error type to `Throwable`, like the Scala `Try` data type, I let the user choose the error type, exactly like `Either`?

Initial results of my experiment were remarkable: just looking at type signatures, I could understand exactly how code was dealing with errors (or not dealing with them). Effect operators precisely reflected error handling behavior in type signatures, and some laws that traditionally had to be checked using libraries like ScalaCheck were now checked statically, at compile time.

So on January 2, 2018, I committed what ended up being a radical departure from the status quo: introducing statically-typed errors into the Scalaz 8 effect type.

Over the months that followed, I worked on polish, optimization, bug fixes, tests, and documentation, and found growing demand to use the data type in production. When it became apparent that Scalaz 8 was a longer-term project, a few ambitious developers pulled the IO data type into a standalone library so they could begin using it in their projects.

I was excited about this early traction, and I didn't want any obstacles to using the data type for users of Scalaz 7.x or Cats, so on June 11, 2018, I decided to pull the project out into a new, standalone project with zero dependencies, completely separate from Scalaz 8.

I chose the name *ZIO*, combining the "Z" from "Scalaz", and the "IO" from "IO monad".

1.5 Contributor Growth

Around this time, the first significant wave of contributors started joining the project, including Regis Kuckaertz, Wiem Zine Elabidine, and Pierre Ricadat (among others)—many new to both open source and functional Scala, although some with deep backgrounds in both.

Through mentorship by me and other contributors, including in some cases weekly meetings and pull request reviews, a whole new generation of open source Scala contributors were born—highly talented functional Scala developers whose warmth positivity and can-do spirit started to shape the community.

ZIO accreted more polish and features to make it easier to build concurrent applications, such as an asynchronous, doubly-back-pressured queue, better error tracking and handling, rigorous finalization, and lower-level resource-safety than bracket.

Although the increased contributions led to an increasingly capable effect type, I personally found that using ZIO was not very pleasant, because of the library's poor type inference.

1.6 Improved Type-Inference

As many functional Scala developers at the time, I had absorbed the prevailing wisdom about how functional programming should be done in Scala, and this meant avoiding subtyping and declaration- site variance. (Indeed, the presence of subtyping in a language does negatively impact type inference, and using declaration-site variance has a couple drawbacks.)

However, because of this mindset, using ZIO required specifying type parameters when calling many methods, resulting in an unforgiving and joyless style of programming, particularly with typed errors. In private, I wrote a small prototype showing that using declaration-site variance could significantly improve type inference, which made me want to implement the feature in ZIO.

At the time, however, ZIO still resided in the Scalaz organization, in a separate repository. I was aware that such a departure from the status quo would be very controversial, so in a private fork, Wiem Zine Elabidine and I worked together on a massive refactoring in our first major collaboration.

On Friday July 20, 2018, we opened the pull request that embraced subtyping and covariance. The results spoke for themselves: nearly all explicit type annotations had been deleted, and although there was still some controversy, it was difficult to argue with the results. With this change, ZIO started becoming pleasant to use, and the extra error type parameter no longer negatively impacted usability, because it could always be inferred and widened seamlessly as necessary.

This experience emboldened me to start breaking other taboos: I started aggressively renaming methods and classes and removing jargon known only to pure

functional programmers. At each step, this created yet more controversy, but also further differentiated ZIO from some of the other options in the landscape, including those in Scalaz 7.x.

From all this turbulent evolution, a new take on functional Scala entered the ZIO community: a contrarian but principled take that emphasizes practical concerns, solving real problems in an accessible and joyful way, using all of Scala, including subtyping and declaration-site variance.

Finally, the project began to feel like the ZIO of today, shaped by a rapidly growing community of fresh faces eager to build a new future for functional programming in Scala.

1.7 Batteries Included

Toward the latter half of 2018, ZIO got compositional scheduling, with a powerful new data type that represents a schedule, equipped with rich compositional operators. Using this single data type, ZIO could either retry effects or repeat them according to near arbitrary schedules.

Artem Pyanykh implemented a blazing fast low-level ring-buffer, which, with the help of Pierre Ricadat, became the foundation of ZIO's asynchronous queue, demonstrating the ability of the ZIO ecosystem to create de novo high-performance JVM structures.

Itamar Ravid, a highly talented Scala developer, joined the ZIO project and added a *Managed* data type encapsulating resources. Inspired by Haskell, *Managed* provided compositional resource safety in a package that supported parallelism and safe interruption. With the help of Maxim Schuwalow, *Managed* has grown to become an extremely powerful data type.

Thanks to the efforts of Raas Ahsan, ZIO unexpectedly got an early version of what would later become `FiberRef`, a fiber-based version of `ThreadLocal`. Then Kai, a wizard-level Scala developer and type astronaut, labored to add compatibility with Cats Effect libraries, so that ZIO users could benefit from all the hard work put into libraries like Doobie, http4s, and FS2.

Thanks to the work of numerous contributors spread over more than a year, ZIO became a powerful solution to building concurrent applications—albeit, one without concurrent streams.

1.8 ZIO Stream

Although Akka Streams provides a powerful streaming solution for Scala developers, it's coupled to the Akka ecosystem and Scala's Future, and doesn't embrace the full compositional power of Scala.

In the functional Scala space, FS2 provides a streaming solution that works with ZIO but it's based on Cats Effect, whose type classes can't benefit from ZIO-specific features.

I knew that a ZIO-specific streaming solution would be more expressive and more type safe, with a lower barrier of entry for existing ZIO users. Given the importance of streaming to modern applications, I decided that ZIO needed its own streaming solution, one unconstrained by the limitations of Cats Effect.

Bringing a new competitive streaming library into existence would be a lot of work, and so when Itamar Ravid volunteered to help, I instantly said yes.

Together, in the third quarter of 2018, Itamar and I worked in secret on *ZIO Stream*, an asynchronous, back-pressured, resource-safe, and compositional stream. Inspired by the remarkable work of Torreborre, as well as work in Haskell on iteratees, the initial release of ZIO Streams delivered a high-performance, composable concurrent streams and sinks, with strong guarantees of resource safety, even in the presence of arbitrary interruption.

We unveiled the design at Scale by the Bay 2018, and since then, thanks to Itamar and his army of capable contributors (including Regis Kuckaertz), ZIO Streams has become one of the highlights ofthe ZIO library—every bit as capable as other streaming libraries, but with much smoother integration with the ZIO effect type and capabilities.

Toward the end of 2018, I decided to focus on the complexity of testing code written using effect systems, which led to the last major revision of the ZIO effect type.

1.9 ZIO Environment

When exploring a contravariant reader data type to model dependencies, I discovered that using intersection types (emulated by the with keyword in Scala 2.x), one could achieve flawless type inference when composing effects with different dependencies, which provided a possible solution to simplifying testing of ZIO applications.

Excitedly, I wrote up a simple toy prototype and shared it with Wiem Zine Elabidine. *"Do you want to help work on this?"* I asked. She said yes, and together we quietly added the third and final type parameter to the ZIO effect type: the environment type parameter.

I unveiled the third type parameter at a now-infamous talk, *The Death of Finally Tagless*, humorously presented with a cartoonish Halloween theme. In this talk, I argued that testability was the primary benefit of the so-called "tagless-final" technique, and that it could be obtained much more simply and in a more teachable way by just "passing interfaces"—the very same solution that object-oriented programmers have used for decades.

As with tagless-final, and under the assumption of discipline, ZIO Environment provided a way to reason about dependencies statically. But unlike tagless-final, it's a joy to use because it fully infers, and doesn't require teaching type classes, category theory, higher-kinded types, and implicits.

Some ZIO users immediately started using ZIO Environment, appreciating the ability to describe dependencies using types without actually passing them. Constructing ZIO environments, however, proved to be problematic—impossible to do generically, and somewhat painful to do even when the structure of the environment was fully known.

A workable solution to these pains would not be identified until almost a year later.

Meanwhile, ZIO continued to benefit from numerous contributions, which added operators improved documentation, improved interop, and improved semantics for core data types.

The next major addition to ZIO was software transactional memory.

1.10 Software Transactional Memory

The first prototype of the Scalaz IO data type included `MVar`, a doubly-back-pressured queue with a maximum capacity of 1, inspired by Haskell's data type of the same name.

I really liked the fact that MVar was already "proven", and could be used to build many other concurrent data structures (such as queues, semaphores, and more).

Soon after that early prototype, however, the talented and eloquent Fabio Labella convinced me that two simpler primitives provided a more orthogonal basis for building concurrency structures:

- *Promise*, a variable data type that can be set exactly one time (but can be awaited on asynchronously and retrieved any number of times);
- *Ref*, a model of a mutable cell that can store any immutable value, with atomic operations for updating the cell.

This early refactoring allowed us to delete `MVar` and provided a much simpler foundation. However, after a year of using these structures, while I appreciated their power, it became apparent to me that they were the "assembly language" of concurrent data structures.

These structures could be used to build lots of other asynchronous concurrent data structures, such as semaphores, queues, and locks, but doing so was extremely tricky, and required hundreds of lines of fairly advanced code.

Most of the complexity stems from the requirement that operations on the data structures must be safely interruptible, without leaking resources or deadlocking.

Moreover, although you can build concurrent structures with `Promise` and `Ref`, you cannot make coordinated changes across two or more such concurrent structures.

The transactional guarantees of structures built with `Promise` and `Ref` are non-compositional: they apply only to isolated data structures, because they are built with `Ref`, which has non-compositional transactional semantics. Strictly speaking, their transactional power is equivalent to actors with mutable state: each actor can safely mutate its own state, but no transactional changes can be made across multiple actors.

Familiar with Haskell's software transactional memory, and how it provides an elegant, compositional solution to the problem of developing concurrent structures, I decided to implement a version for ZIO with the help of my partner-in-crime Wiem Zine Elabidine, which we presented at Scalar Conf in April 2019.

Soon after, Dejan Mijic, a fantastic and highly motivated developer with a keen interest in high-performance, concurrency, and distributed systems, joined the ZIO STM team. With my mentorship, Dejan helped make STM stack-safe for transactions of any size, added several new STM data structures, dramatically improved the performance of existing structures, and implemented retry-storm protection for supporting large transactions on hotly contested transactional references.

ZIO STM is the only STM in Scala with these features, and although the much older Scala STM is surely production-worthy, it doesn't integrate well with asynchronous and purely functional effect systems built using fiber-based concurrency.

The next major feature in ZIO would address a severe deficiency that had never been solved in the Scala ecosystem: the extreme difficulty of debugging async code, a problem present in Scala's Future for more than a decade.

1.11 Execution Traces

Previously in presenting ZIO to new non-pure functional programmers (the primary audience for ZIO), I had received the question: how do we debug ZIO code?

The difficulty stems from the worthless nature of stack traces in highly asynchronous programming. Stack traces only capture the call stack, but in Future and ZIO and other heavily async environments, the call stack mainly shows you the "guts" of the execution environment, which is not very useful for troubleshooting errors.

I had thought about the problem and had become convinced it would be possible to implement async execution traces using information reconstructed from the

call stack, so I began telling people we would soon implement something like this in ZIO.

I did not anticipate just how soon this would happen.

Kai came to me with an idea to do execution tracing in a radically different way than I imagined: by dynamically parsing the bytecode of class files. Although my recollection is a bit hazy, it seemed mere days before Kai had whipped up a prototype that seemed extremely promising, so I offered my assistance on hammering out the details of the full implementation, and we ended up doing a wonderful joint talk in Ireland to launch the feature.

Sometimes I have a tendency to focus on laws and abstractions, but seeing the phenomenally positive response to execution tracing was a good reminder to stay focused on the real world pains that developers have.

1.12 Summer 2019

Beginning in the summer of 2019, ZIO began seeing its first significant commercial adoption, which led to many feature requests and bug reports, and much feedback from users.

The summer saw many performance improvements, bug fixes, naming improvements, and other tweaks to the library, thanks to Regis Kuckaertz and countless other contributors.

Thanks to the work of the ever-patient Honza Strnad and others, `FiberRef` evolved into its present-day form, which is a much more powerful, fiber-aware version of ThreadLocal—but one which can undergo specified transformations on forks, and merges on joins.

I was very pleased with these additions. However, as ZIO grew, the automated tests for ZIO were growing too, and they became an increasing source of pain across Scala.js, JVM, and Dotty (the test runners at the time did not natively support Dotty).

So in the summer of 2019, I began work on a purely functional testing framework, with the goal of addressing these pains, the result of which was ZIO Test.

1.13 ZIO Test

Testing functional effects inside a traditional testing library is painful: there's no easy way to run effects, provide them with dependencies, or integrate with the host facilities of the functional effect system (using retries, repeats, and so forth).

I wanted to change that with a small, compositional library called ZIO Test, whose design I had beenthinking about since even before ZIO existed.

Like the ground-breaking Specs2 before it, ZIO Test embraced a philosophy of tests as values, although ZIO Test retained a more traditional tree-like structure for specs, which allows nesting tests inside test suites, and suites inside other suites.

Early in the development of ZIO Test, the incredible and extremely helpful Adam Fraser joined the project as a core contributor. Instrumental to fleshing out, realizing, and greatly extending the vision for ZIO Test, Adam has since become the lead architect and maintainer for the project, and co-author of this book.

Piggybacking atop ZIO's powerful effect type, ZIO Test was implemented in comparatively few lines of code: concerns like retrying, repeating, composition, parallel execution, and so forth, were already implemented in a principled, performant, and type-safe way.

Indeed, ZIO Test also got a featherweight alternative to ScalaCheck based on ZIO Streams, since a generator of a value can be viewed as a stream. Unlike ScalaCheck, the ZIO Test generator has auto-shrinking baked in, inspired by the Haskell Hedgehog library; and it correctly handles filters on shrunk values and other edge case scenarios that ScalaCheck did not handle.

Toward the end of 2018, after nearly a year of real world usage, the ZIO community had been hard at work on solutions to the problem of making dynamic construction of ZIO environments easier.

This work directly led to the creation of *ZLayer*, the last major data type added to ZIO.

1.14 ZLayer

Two very talented Scala developers, Maxim Schuwalow and Piotr Gołębiewski, jointly worked on a ZIO Macros project, which, among other utilities, provided an easier way to construct larger ZIO environments from smaller pieces. This excellent work was independently replicated in Netflix's highly-acclaimed Polynote by Scala engineer Jeremy Smith, in response to the same pain.

At Functional Scala 2019, several speakers presented on the pain of constructing ZIO Environments, which convinced me to take a hard look at the problem. Taking inspiration from an earlier attempt by Piotr, I created two new data types, `Has` and `ZLayer`.

`Has` can be thought of as a type-indexed heterogeneous map, which is type safe, but requires access to compile-time type tag information. `ZLayer` can be thought of as a more powerful version of Java and Scala constructors, which can build multiple services in terms of their dependencies.

Unlike constructors, ZLayer dependency graphs are ordinary values, built from other values using composable operators, and ZLayer supports resources, asyn-

chronous creation and finalization, retrying, and other features not possible with constructors.

ZLayer provided a very clean solution to the problems developers were having with ZIO Environment—not perfect, mind you, and I don't think any solution prior to Scala 3 can be perfect (every solution in the design space has different tradeoffs). This solution became even better when the excellent consultancy Septimal Mind donated Izumi Reflect to the ZIO organization.

The introduction of ZLayer was the last major change to any core data type in ZIO. Since then, although streams has seen some evolution, the rest of ZIO was quite stable.

Yet despite the stability, until August 2020, there was still one major unresolved issue at the very heart of the ZIO runtime system: a full solution to the problem of structured concurrency.

1.15 Structured Concurrency

Structured concurrency is a paradigm that provides strong guarantees around the lifespans of operations performed concurrently. These guarantees make it easier to build applications that have stable, predictable resource utilization.

Since I have long been a fan of Haskell structured concurrency (via Async and related), ZIO was the first effect system to support structured concurrency in numerous operations:

- By default, interrupting a fiber does not return until the fiber has been interrupted and all its finalizers executed.
- By default, timing out an effect does not return until the effect being timed out has been interrupted and all its finalizers executed.
- By default, when executing effects in parallel, if one of them fails, the parallel operation will not continue until all sibling effects have been interrupted.
- Etc.

Some of these design decisions were highly contentious and have not been implemented in other effect systems until recently (if at all).

However, there was one notable area where ZIO did not provide default structured concurrency: whenever an effect was forked (launched concurrently to execute on a new fiber), the lifespan of the executing effect was unconstrained.

Solving this problem turned out to require major surgery to the ZIO internal runtime system (which is a part of ZIO that few contributors understand completely).

In the end, we solved the problem in a satisfactory way, making ZIO the only effect system to fully support structured concurrency. But it required learning

from real world feedback and prototyping no less than 5 completely different solutions to the problem.

So after three years of development, on August 3rd, 2020, ZIO 1.0 was released live in an online Zoom-hosted launch party that brought together and paid tribute to contributors and users across the ZIO ecosystem. We laughed, we chatted, I rambled for a while, and we toasted to users, contributors, and the past and future of ZIO.

1.16 Why ZIO

ZIO is a new library for concurrent programming. Using features of the Scala programming language, ZIO helps you build efficient, resilient, and concurrent applications that are easy to understand and test, and which don't leak resources, deadlock, or lose errors.

Used pervasively across an application, ZIO simplifies many of the challenges of building modern applications:

- **Concurrency**. Using an asynchronous fiber-based model of concurrency that never blocks threads or deadlocks, ZIO can run thousands or millions of virtual threads concurrently.
- **Efficiency**. ZIO automatically cancels running computations when the result of the computations are no longer necessary, providing global application efficiency for free.
- **Error Handling**. ZIO lets you track errors statically, so the compiler can tell you which code has handled its errors, and which code can fail, including how it can fail.
- **Resource-Safety**. ZIO automatically manages the lifetime of resources, safely acquiring them and releasing them even in the presence of concurrency and unexpected errors.
- **Streaming**. ZIO has powerful, efficient, and concurrent streaming that works with any source of data, whether structured or unstructured, and never leaks resources.
- **Troubleshooting**. ZIO captures all errors, including parallel and finalization errors, with detailed execution traces and suspension details that make troubleshooting applications easy.
- **Testability**. With *dependency inference*, ZIO makes it easy to code to interfaces, and ships with testable clocks, consoles, and other core system modules.

ZIO frees application developers to focus on business logic, and fully embraces the features of the Scala programming languages to improve productivity, testability, and resilience.

Since it's 1.0 release in August of 2020, ZIO has sparked a teeming ecosystem of ZIO-compatible libraries that provide support for GraphQL, persistence, REST

APIs, microservices, and much more.

1.17 ZIO Alternatives

ZIO is not the only choice for concurrent programming in Scala. In addition to libraries and frameworks in the Java ecosystem, Scala programmers have their choice of several competing solutions with first-class support for the Scala programming language:

- **Akka**. Akka is a toolkit for building reactive, concurrent, and distributed applications.
- **Monix**. Monix is a library for composing asynchronous, event-based programs.
- **Cats Effect**. Cats Effect is a purely functional runtime system for Scala.

Akka is older and more mature, with a richer ecosystem and more production usage, but ZIO provides compositional transactionality, resource-safety, full testability, better diagnostics, and greatly improved resilience via compile-time error analysis.

Monix is more focused on reactive programming, and less focused on concurrent programming, but to the extent it overlaps with ZIO, ZIO has significantly more expressive power.

Cats Effect is more focused on so-called *tagless-final* type classes than concurrent programming, and while it has some concurrent features, it is much less expressive than ZIO, without any support for compositional transactionality.

Beyond just technical merits, I think there are compelling reasons for Scala developers to take a serious look at ZIO:

- **Wonderful Community**. I'm amazed at all the talent, positivity, and mentorship seen in the ZIOcommunity, as well as the universal attitude that we are all on the same team—there is no *your code*, or *my code*, just *our code*.

2. **History of Innovation**. ZIO has been the first effect type with proper thread pool locking, typed errors, environment, execution tracing, fine-grained interruption, structured concurrency, and so much more. Although inspired by Haskell, ZIO has charted its own course and become what manybelieve to be the leading effect system in Scala.
3. **A Bright Future**. ZIO took three years to get right, because I believed the foundations had to be extremely robust to support a compelling next-generation ecosystem. If early libraries like Caliban, ZIO Redis, ZIO Config, ZIO gRPC, and others are any indication, ZIO will continue to become a hotbed of exciting new developments for the Scala programming language.

Concurrent programming in Scala was never this much fun, correct-by-construction, or productive.

1.18 Zionomicon

In your hands, you have Zionomicon, a comprehensive book lovingly crafted by myself and Adam Fraser with one goal: to turn you into a wizard at building modern applications.

Through the course of this book, you will learn how to build cloud-ready applications that are responsive, resilient, elastic, and event-driven.

They will be low-latency and globally efficient.

They will not block threads, leak resources, or deadlock.

They will be checked statically at compile-time by the powerful Scala compiler.

They will be fully testable, straightforward to troubleshoot with extensive diagnostics.

They will deal with errors in a principled and robust fashion, surviving transient failures, and handling business errors according to requirements.

In short, our goal with this book is to help you become a programmer of extraordinary power, by leveraging both the Scala programming language and the power of functional composition.

Congratulations on taking your first step toward mastering the dark art of ZIO!

Chapter 2

Essentials: First Steps With ZIO

Congratulations on taking your first step in mastering ZIO!

ZIO will help you build modern applications that are concurrent, resilient, and efficient, as well as easy to understand and test. But learning ZIO requires thinking about software in a whole new way—a way that comes from *functional programming*.

This chapter will teach you the critical theory you need to understand and build ZIO applications.

We will start by introducing the core data type in ZIO, which is called a *functional effect type*, and define functional effects as *blueprints* for concurrent workflows. We will learn how to combine effects sequentially, and see how this allows us to refactor legacy code to ZIO.

We will discuss the meaning of each of the type parameters in ZIO's core data type, particularly the error type and the environment type, which are features unique to ZIO. We will compare `ZIO` to the `Future` data type in the Scala standard library, to clarify the concepts we introduce.

We will see how ZIO environment lets us leverage the testable services built into ZIO for interacting with time, the console, and system information (among others). Finally, we'll see how recursive ZIO effects allow us to loop and perform other control flow operations.

By the end of this chapter, you will be able to write basic programs using `ZIO`, including those that leverage environmental effects and custom control flow operators, and you will be able to refactor legacy code to `ZIO` by following some simple guidelines.

2.1 Functional Effects As Blueprints

The core data type in the ZIO library is `ZIO[R, E, A]`, and values of this type are called *functional effects*.

A functional effect is a kind of *blueprint for a concurrent workflow*, as illustrated in Figure 1. The blueprint is purely descriptive in nature, and must be executed in order to observe any side-effects, such as interaction with a database, logging, streaming data across the network, or accepting a request.

Figure 1

A functional effect of type `ZIO[R, E, A]` requires you to supply a value of type `R` if you want to execute the effect (this is called the *environment* of the effect), and when it is executed, it may either fail with a value of type `E` (the *error type*, or succeed with a value of type `A` (the *success type*).

We will talk more about each of these type parameters shortly. But first, we need to understand what it means for an effect to be a *blueprint*.

In traditional procedural programming, we are used to each line of our code directly interacting with the outside world. For example, consider the following snippet:

```
val goShopping: Unit = {
  println("Going to the grocery store")
}
```

As soon as Scala computes the unit value for the `goShopping` variable, the application will immediately print the text "Going to the grocery store" to the console.

This is an example of *direct execution*, because in constructing a value, our program directly interacts with the outside world.

This style of programming is called *procedural programming*, and is familiar to almost all programmers, since most programming languages are procedural in nature.

Procedural programming is convenient for simple programs. But when we write our programs in this style, *what* we want to do (going to the store) becomes tangled with *how* we want to do it (going to the store *now*).

This tangling can lead to lots of boilerplate code that is difficult to understand and test, painful to change, and fraught with subtle bugs that we won't discover until production.

For example, suppose we don't actually want to go to the grocery store now but in an hour from now. We might try to implement this new feature by using a `ScheduledExecutorService`:

```
import java.util.concurrent.{ Executors, ScheduledExecutorService }
import java.util.concurrent.TimeUnit._

val scheduler: ScheduledExecutorService =
  Executors.newScheduledThreadPool(1)

scheduler.schedule(
  new Runnable { def run: Unit = goShopping },
  1,
  HOURS
)
scheduler.shutdown()
```

In this program, we create an executor, schedule `goShopping` to be executed in one hour, and then shut down the scheduler when we are done. (Don't worry if you don't understand everything that is going on here. We will see that ZIO has much easier ways of doing the same thing!)

Not only does this solution involve boilerplate code that is difficult to understand and test, and painful to change, but it also has a subtle bug!

Because `goShopping` is directly executed, rather than being a blueprint for a workflow, "Going to the grocery store" will be printed to the console as soon as `goShopping` is loaded by the JVM. So we will be going to the grocery store now instead of an hour from now!

In fact, the only thing we have scheduled to be executed in an hour is returning the `Unit` value of `goShopping`, which doesn't do anything at all.

In this case, we can solve the problem by defining `goShopping` as a `def` instead of a `val` to defer its evaluation until later. But this approach is fragile and error prone, and forces us to think carefully about when each statement in our program will be evaluated, which is no longer the order of the statements.

We also have to be careful not to accidentally evaluate a statement too early. We might assign it to a value or put it into a data structure, which could cause premature evaluation.

It is as if we want to talk to our significant other about going shopping, but as soon as we mention the word "groceries", they are already at the door!

The solution to this problem (and most problems in concurrent programming) is to make the statements in our program *values* that *describe* what we want to do. This way, we can separate *what we want to do* from *how we want to do it.*

The following snippet shows what this looks like with ZIO:

```
import zio._

val goShopping =
  ZIO.effect(println("Going to the grocery store"))
```

Here we are using `effect` constructor to build the `goShopping` functional effect. The effect is a blueprint that *describes* going to the store, but doesn't actually do anything right now. (To prove this to yourself, try evaluating the code in the Scala REPL!)

In order to go to the store, we have to *execute* the effect, which is clearly and forcibly separated from defining the effect, allowing us to untangle these concerns and simplifying code tremendously.

With `goShopping` defined this way, we can now describe *how* independent from *what*, which allows us to solve complex problems compositionally, by using operations defined on ZIO effects.

Using the `delay` operator that is defined on all ZIO effects, we can take `goShopping`, and transform it into a new effect, which will go shopping an hour from now:

```
import zio.clock._
import zio.duration._

val goShoppingLater =
  goShopping.delay(1.hour)
```

Notice how easy it was for us to reuse the original effect, which specified *what*, to produce a new effect, which also specified *when*. We built a solution to a more complex problem by transforming a solution to a simpler problem.

Thanks to the power of describing workflows as ordinary immutable values, we never had to worry about how `goShopping` was defined or about evaluating it too early. Also, the value returned by the `delay` operator is just another description, so we can easily use it to build even more sophisticated programs in the same way.

In ZIO, every `ZIO` effect is just a description—a blueprint for a concurrent workflow. As we write our program, we create larger and more complex blueprints that come closer to solving our business problem. When we are done and have an effect that describes everything we need to do, we hand it off to the ZIO runtime, which executes the blueprint and produces the result of the program.

So how do we actually run a `ZIO` effect? The easiest way is to extend the `App` trait and implement the `run` method, as shown in the following snippet:

```
import zio._

object GroceryStore extends App {
  def run(args: List[String]) =
    goShopping.exitCode
}
```

The `run` function requires that we return an `ExitCode`, which describes the exit value to use when the process terminates. The `exitCode` method defined on

effects is a convenience method that translates all successes to `ExitCode(0)` and failures to an `ExitCode(1)`.

As you are experimenting with ZIO, extending `App` and implementing your own program logic in the `run` method is a great way to see the output of different programs.

2.2 Sequential Composition

As discussed above, ZIO effects are blueprints for describing concurrent workflows, and we build more sophisticated effects that come closer to solving our business problem by transforming and combining smaller, simpler effects.

We saw how the `delay` operator could be used to transform one effect into another effect whose execution is delayed into the future. In addition to `delay`, ZIO has dozens of other powerful operators that transform and combine effects to solve common problems in modern application development.

We will learn about most of these operators in subsequent chapters, but one of the most important operators that we need to introduce is called `flatMap`.

The `flatMap` method of ZIO effects represents sequential composition of two effects, allowing us to create a second effect based on the output of the first effect.

A simplified type signature for `flatMap` looks something like this:

```
trait ZIO[R, E, A] {
  ...
  def flatMap[B](andThen: A => ZIO[R, E, B]): ZIO[R, E, B] = ...
  ...
}
```

In effect, `flatMap` says, "run the first effect, then run a second effect that depends on the result of the first one". Using this sequential operator, we can describe a simple workflow that readers user input and then displays the input back to the user, as shown in the following snippet:

```
import scala.io.StdIn

val readLine =
  ZIO.effect(StdIn.readLine())

def printLine(line: String) =
  ZIO.effect(println(line))

val echo =
  readLine.flatMap(line => printLine(line))
```

Notice how what we print on the console depends on what we read from the console: so we are doing two things in sequence, and the second thing that we do depends on the value produced by the first thing we do.

The `flatMap` operator is fundamental because it captures the way statements are executed in a procedural program: later statements depend on results computed by previous statements, which is exactly the relationship that `flatMap` describes.

For reference, here is the above program written in a procedural style:

```
val line = Console.readLine
Console.println(line)
```

This relationship between procedural programming and the `flatMap` operator is so precise, we can actually translate any procedural program into ZIO by wrapping each statement in a constructor like `ZIO.effect` and then gluing the statements together using `flatMap`.

For example, let's say we have the procedural program shown in the following snippet:

```
val data = doQuery(query)
val response = generateResponse(data)
writeResponse(response)
```

We can translate this program into ZIO as follows:

```
ZIO.effect(doQuery(query)).flatMap(data =>
  ZIO.effect(generateResponse(data)).flatMap(response =>
    ZIO.effect(writeResponse(response))
  )
)
```

Although a straightforward transformation, once you exceed two or three `flatMap` operations in a row, the nesting of the code becomes somewhat hard to follow. Fortunately, Scala has a feature called *for comprehensions*, which allow us to express sequential composition in a way that looks like procedural programming.

In the next section, we'll explore *for comprehensions* at length.

2.2.1 For Comprehensions

Using *for comprehensions*, we can take the following Scala snippet:

```
readLine.flatMap(line => printLine(line))
```

and rewrite it into the following *for comprehension*:

```
import zio._

val echo =
  for {
```

```
    line <- readLine
    _    <- printLine(line)
  } yield ()
```

As you can see from this short snippet, there is no nesting, and each line in the comprehension looks similar to a statement in procedural programming.

For comprehensions have the following structure:

1. They are introduced by the keyword `for`, followed by a code block, and terminated by the keyword `yield`, which is followed by a single parameter, representing the success value of the effect.
2. Each line of the *for comprehension* is written using the format `result <- effect`, where `effect` returns an effect, and `result` is a variable that will hold the success value of the effect. If the result of the effect is not needed, then the underscore may be used as the variable name.

A *for comprehension* with `n` lines is translated by Scala into `n - 1` calls to `flatMap` methods on the effects, followed by a final call to a `map` method on the last effect.

So, for example, if we have the following *for comprehension*:

```
for {
  x <- doA
  y <- doB(x)
  z <- doC(x, y)
} yield x + y + z
```

Then Scala will translate it into the following code:

```
doA.flatMap(x =>
  doB(x).flatMap(y =>
    doC(x, y).map(z => x + y + z)))
```

Many Scala developers find that *for comprehensions* are easier to read than long chains of nested calls to `flatMap`. In this book, except for very short snippets, we will prefer *for comprehensions* over explicit calls to `flatMap`.

2.3 Other Sequential Operators

Sequential composition is so common when using functional effects, ZIO provides a variety of related operators for common needs.

The most basic of these is `zipWith`, which combines two effects sequentially, merging their two results with the specified user-defined function.

For example, if we have two effects that prompt for the user's first name and last name, then we can use `zipWith` to combine these effects together sequentially, merging their results into a single string:

```
val firstName =
  ZIO.effect(StdIn.readLine("What is your first name?"))

val lastName =
  ZIO.effect(StdIn.readLine("What is your last name"))

val fullName =
  firstName.zipWith(lastName)((first, last) => s"$first $last")
```

The `zipWith` operator is less powerful than `flatMap`, because it does not allow the second effect to depend on the first, even though the operator still describes sequential, left-to-right composition.

Other variations include `zip`, which sequentially combines the results of two effects into a tuple of their results; `zipLeft`, which sequentially combines two effects, returning the result of the first; and `zipRight`, which sequentially combines two effects returning the result of the second.

Occasionally, you will see `<*` used as an alias for `zipLeft`, and `*>` as an alias for `zipRight`. These operators are particularly useful to combine a number of effects sequentially when the result of one or more of the effects are not needed.

For example, in the following snippet, we sequentially combine two effects, returning the `Unit` success value of the right hand effect:

```
val helloWorld =
  ZIO.effect(print("Hello, ")) *> ZIO.effect(print("World!\n"))
```

This is useful because even while `Unit` is not a very useful success value, a tuple of unit values is even less useful!

Another useful set of sequential operators is `foreach` and `collectAll`.

The `foreach` operator returns a single effect that describes performing an effect for each element of a collection in sequence. It's similar to a *for loop* in procedural programming, which iterates over values, processes them in some fashion, and collects the results.

For example, we could create an effect that describes printing all integers between `1` and `100` like this:

```
val printNumbers =
  ZIO.foreach(1 to 100) { n =>
    printLine(n.toString)
  }
```

Similarly, `collectAll` returns a single effect that collects the results of a whole collection of effects. We could use this to collect the results of a number of printing effects, as shown in the following snippet:

```
val prints =
  List(
```

```
    printLine("The"),
    printLine("quick"),
    printLine("brown"),
    printLine("fox")
  )

val printWords =
  ZIO.collectAll(prints)
```

With just what you have learned so far, you can take any procedural program and translate it into a ZIO program by wrapping statements in effect constructors and combining them with `flatMap`.

If that is all you do, you won't be taking advantage of all the features that ZIO has to offer, but it's a place to start, and it can be a useful technique when migrating legacy code to ZIO.

2.4 ZIO Type Parameters

We said before that a value of type `ZIO[R, E, A]` is a functional effect that requires an environment `R` and may either fail with an `E` or succeed with an `A`.

Now that we understand what it means for a `ZIO` effect to be a blueprint for a concurrent workflow, and how to combine effects, let's talk more about each of the ZIO type parameters:

- `R` is the environment required for the effect to be executed. This could include any dependencies the effect has, for example access to a database or a logging service, or an effect might not require any environment, in which case, the type parameter will be `Any`.
- `E` is type of value that the effect can fail with. This could be `Throwable` or `Exception`, but it could also be a domain-specific error type, or an effect might not be able to fail at all, in which case the type parameter will be `Nothing`.
- `A` is the type of value that the effect can succeed with. It can be thought of as the return value or output of the effect.

A helpful way to understand these type parameters is to imagine a `ZIO` effect as a function `R => Either[E, A]`. This is not actually the way `ZIO` is implemented (this definition wouldn't allow us to write concurrent, async or resource-safe operators, for example), but it is a useful mental model.

The following snippet of code defines this toy model of a ZIO effect:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A])
```

As you can see from this definition, the `R` parameter is an *input* (in order to execute the effect, you must supply a value of type `R`), while the `E` and `A`

parameters are *outputs*. The input is declared to be *contravariant*, and the outputs are declared to be *covariant*.

For a more detailed discussion of variance, see the appendix. Otherwise, just know that Scala's variance annotations *improve type inference*, which is why ZIO uses them.

Let's see how we can use this mental model to implement some basic constructors and operators:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =>
  def map[B](f: A => B): ZIO[R, E, B] =
    ZIO(r => self.run(r).map(f))
  def flatMap[R1 <: R, E1 >: E, B](
    f: A => ZIO[R1, E1, B]
  ): ZIO[R1, E1, B] =
    ZIO(r => self.run(r).fold(ZIO.fail(_), f).run(r))
}

object ZIO {
  def effect[A](a: => A): ZIO[Any, Throwable, A] =
    ZIO(_ => try Right(a) catch { case t : Throwable => Left(t) })
  def fail[E](e: => E): ZIO[Any, E, Nothing] =
    ZIO(_ => Left(e))
}
```

The `ZIO.effect` method wraps a block of code in an effect, converting exceptions into `Left` values, and successes into `Right` values. Notice that the parameter to `ZIO.effect` is *by name* (using the `=> A` syntax), which prevents the code from being evaluated eagerly, allowing ZIO to create a value that *describes* execution.

We also implemented the `flatMap` operator previously discussed, which allows us to combine effects sequentially. The implementation of `flatMap` works as follows:

1. It first runs the original effect with the environment `R1` to produce an `Either[E, A]`.
2. If the original effect fails with a `Left(e)`, it immediately returns this failure as a `Left(e)`.
3. If the original effect succeeds with a `Right(a)`, it calls `f` on that `a` to produce a new effect. It then runs that new effect with the required environment `R1`.

As discussed above, ZIO effects aren't actually implemented like this, but the basic idea of executing one effect, obtaining its result, and then passing it to the next effect is an accurate mental model, and it will help you throughout your time working with ZIO.

We will learn more ways to operate on successful `ZIO` values soon, but for now lets focus on the error and environment types to build some intuition about

them since they may be less familiar.

2.4.1 The Error Type

The error type represents the potential ways that an effect can fail. The error type is helpful because it allows us to use operators (like `flatMap`) that work on the success type of the effect, while deferring error handling until higher-levels. This allows us to concentrate on the "happy path" of the program and handle errors at the right place.

For example, say we want to write a simple program that gets two numbers from the user and multiplies them:

```
import zio._

lazy val readInt: ZIO[Any, NumberFormatException, Int] =
  ???

lazy val readAndSumTwoInts: ZIO[Any, NumberFormatException, Int] =
  for {
    x <- readInt
    y <- readInt
  } yield x * y
```

Notice that `readInt` has a return type of `ZIO[Any, NumberFormatException, Int]`, indicating that it does not require any environment and may either succeed with an integer (if the user enters a response that can be parsed into a valid integer) or fail with a `NumberFormatException`.

The first benefit of the error type is that we know how this function can fail just from its signature. We don't know anything about the implementation of `readInt`, but just looking at the type signature we know that it can fail with a `NumberFormatException`, and can't fail with any other errors. This is very powerful because we know exactly what kind of errors we potentially have to deal with, and we never have to resort to "defensive programming" to handle unknown errors.

The second benefit is that we can operate on the results of effects assuming they are successful, deferring error handling until later. If either `readInt` call fails with a `NumberFormatException`, then `readAndSumTwoInts` will also fail with the exception, and abort the summation. This bookkeeping is handled for us automatically. We can multiply `x` and `y` directly and never have to deal explicitly with the possibility of failure. This defers error handling logic to the caller, which can retry, report, or defer handling even higher.

Being able to see how an effect can fail and to defer errors to a higher level of an application is useful, but at some point, we need to be able to handle some or all errors.

To handle errors with our toy model of ZIO, let's implement an operator called `foldM` that will let us perform one effect if the original effect fails, and another one if it succeeds:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =>
  def foldM[R1 <: R, E1, B](
    failure: E => ZIO[R1, E1, B],
    success: A => ZIO[R1, E1, B]
  ): ZIO[R1, E1, B] =
    ZIO(r => self.run(r).fold(failure, success).run(r))
}
```

The implementation is actually quite similar to the one we walked through above for `flatMap`. We are just using the `failure` function to return a new effect in the event of an error, and then running that effect.

One of the most useful features of the error type is being able to specify that an effect *cannot fail at all*, perhaps because its errors have already been caught and handled.

In ZIO, we do this by specifying `Nothing` as the error type. Since there are no values of type `Nothing`, we know that if we have an `Either[Nothing, A]` it must be a `Right`. We can use this to implement error handling operators that let us statically prove that an effect can't fail because we have handled all errors.

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =>
  def fold[B](failure: E => B, success: A => B): ZIO[R, Nothing, B] =
    ZIO(r => Right(self.run(r).fold(failure, success)))
}
```

2.4.2 The Environment Type

Now that we have some intuition around the error type, let's focus on the environment type.

We can model effects that don't require any environment by using `Any` for the environment type. After all, if an effect requires a value of type `Any`, then you could run it with `()` (the unit value), `42`, or any other value. So an effect that can be run with a value of any type at all is actually an effect that doesn't need any specific kind of environment.

The two fundamental operational of working with the environment are accessing the environment (e.g. getting access to a database to do something with it) and providing the environment (providing a database service to an effect that needs one, so it doesn't need anything else).

We can implement this in our toy model of ZIO as shown in the following snippet:

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A]) { self =>
  def provide(r: R): ZIO[Any, E, A] =
```

```
    ZIO(_ => self.run(r))
}

object ZIO {
  def environment[R]: ZIO[R, Nothing, R] =
    ZIO(r => Right(r))
}
```

As you can see, the `provide` operator returns a new effect that doesn't require any environment. The `environment` constructor creates a new effect with a required environment type, and just passes through that environment as a success value. This allows us to access the environment and work with it using other operators like `map` and `flatMap`.

2.5 ZIO Type Aliases

With its three type parameters `ZIO` is extremely powerful. We can use the environment type parameter to propagate information downward in our program (databases, connection pools, configuration, and much more), and we can use the error and success type parameters to propagate information upward.

In the most general possible case, programs need to propagate dependencies down, and return results up, and ZIO provides all of this in a type-safe package.

But sometimes, we may not need all of this power. We may know that an application doesn't require an external environment, or that it can only fail with a certain type of errors, or that it can't fail at all.

To simplify these cases, ZIO comes with a number of useful type aliases. You never have to use these type aliases if you don't want to. You can always just use the full ZIO type signature. But these type aliases are used frequently in ZIO code bases so it is helpful to be familiar with them, and they can make your code more readable if you choose to use them.

The key type aliases are:

```
type IO[+E, +A]   = ZIO[Any, E, A]
  type Task[+A]     = ZIO[Any, Throwable, A]
  type RIO[-R, +A]  = ZIO[R, Throwable, A]
  type UIO[+A]      = ZIO[Any, Nothing, A]
  type URIO[-R, +A] = ZIO[R, Nothing, A]
```

Here is a brief description of each type alias to help you remember what they are for:

- `IO[E, A]` - An effect that does not require any environment, may fail with an `E`, or may succeed with an `A`
- `Task` - An effect that does not require any environment, may fail with a `Throwable`, or may succeed with an `A`

- `RIO` - An effect that requires an environment of type `R`, may fail with a `Throwable`, or may succeed with an `A`.
- `UIO` - An effect that does not require any environment, cannot fail, and succeeds with an `A`
- `URIO[R, A]` - An effect that requires an environment of type `R`, cannot fail, and may succeed with an `A`.

Several other data types in ZIO and other libraries in the ZIO ecosystem use similar type aliases, so if you are familiar with these you will be able to pick those up quickly, as well.

ZIO comes with companion objects for each of these type aliases, so you can call static methods with these type aliases the same way you would on `ZIO` itself.

For example, these two definitions are equivalent and are both valid syntax:

```
import zio._

val first: ZIO[Any, Nothing, Unit] =
  ZIO.effectTotal(println("Going to the grocery store"))
val second: UIO[Unit] =
  UIO.effectTotal(println("Going to the grocery store"))
```

ZIO strives for excellent type inference across the board, but in rare cases where type parameters have to be specified explicitly, the constructors on the companion object can require fewer type parameters (e.g. constructors on `UIO` have no error parameter) so they can improve type inference and ergonomics.

2.6 Comparison to Future

We can clarify what we have learned so far by comparing `ZIO` with `Future` from the Scala standard library.

We will discuss other differences between `ZIO` and `Future` later in this book when we discuss concurrency, but for now there are three primary differences to keep in mind.

2.6.1 A Future Is A Running effect

Unlike a functional effect like `ZIO`, a `Future` models a running effect. To go back to our example from the beginning of the chapter, consider this snippet:

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

val goShopping: Future[Unit] =
  Future(println("Going to the grocery store"))
```

Just like our original example, as soon as `goShopping` is defined this effect will begin executing. `Future` does not suspend evaluation of code wrapped in it.

Because of this tangling between the *what* and the *how*, we don't have much power when using `Future`. For example, it would be nice to be able to define a `delay` operator on `Future`, just like we have for `ZIO`. But we can't do that because it would be a method on `Future`, and if we have a `Future`, then it is already running, so it's too late to delay it.

Similarly, we can't retry a `Future` in the event of failure, like we can for `ZIO`, because a `Future` isn't a blueprint for doing something—it's an executing computation. So if a `Future` fails, there is nothing else to do. We can only retrieve the failure.

In contrast, since a `ZIO` effect is a blueprint for a concurrent workflow, if we execute the effect once and it fails, we can always try executing it again, or executing it as many times as we would like.

One case in which this distinction is particularly obvious is the persistent requirement that you have an implicit `ExecutionContext` in scope whenever you call methods on `Future`.

For example, here is the signature of `Future#flatMap`, which like `flatMap` on `ZIO`, allows us to compose sequential effects:

```
import scala.concurrent.ExecutionContext

trait Future[+A] {
  def flatMap[B](f: A => Future[B])(
    implicit ec: ExecutionContext
  ): Future[B]
}
```

`Future#flatMap` requires an `ExecutionContext` because it represents a running effect, so we need to provide the `ExecutionContext` on which this subsequent code should be immediately run.

As discussed before, this conflates *what* should be done with *how* it should be done. In contrast, none of the code involving `ZIO` we have seen requires an `Executor` because it is just a blueprint.

ZIO blueprints can be run on any `Executor` we want, but we don't have to specify this until we actually run the effect (or, later we will see how you can "lock" an effect to run in a specific execution context, for those rare cases where you need to be explicit about this).

2.6.2 Future Has An Error Type Fixed To Throwable

`Future` has an error type fixed to `Throwable`. We can see this in the signature of the `Future#onComplete`:

```
import scala.util.Try

trait Future[+A] {
  def onComplete[B](f: Try[A] => B): Unit
}
```

The result of a `Future` can either be a `Success` with an `A` value, or a `Failure` with a `Throwable`. When working with legacy code that can fail for any `Throwable`, this can be convenient, but it has much less expressive power than a polymorphic error type.

First, we don't know by looking at the type signature how or even if an effect can fail. Consider the multiplication example we looked at when discussing the `ZIO` error type implemented with `Future`:

```
def parseInt: Future[Int] =
  ???
```

Notice how we had to define this as a `def` instead of a `val` because a `Future` is a running effect. So if we defined it as a `val`, we would immediately be reading and parsing an input from the user. Then when we used `parseInt`, we'd always get back the same value, instead of prompting the user for a new value and parsing that.

Putting this aside, we have no idea how this future can fail by looking at the type signature. Could it return a `NumberFormatException` from parsing? Could it return an `IOException`? Could it not fail at all because it handles its own errors, perhaps by retrying until the user entered a valid integer? We just don't know, not unless we dig into the code and study it at length.

This makes it much harder for developers who call this method, because they don't know what type of errors can occur, so to be safe they need to do "defensive programming" and handle any possible `Throwable`.

This problem is especially annoying when we handle all possible failure scenarios of a `Future`, but nothing changes about the type.

For example, we can handle `parseInt` errors by using the `Future` method `fallbackTo`:

```
import scala.concurrent.Future

def parseIntOrZero: Future[Int] =
  parseInt.fallBackTo(Future.successful(0))
```

Here `parseIntOrZero` cannot fail, because if `parseInt` fails, we will replace it with a successful result of `0`. But the type signature doesn't tell us this. As far as the type signature is concerned, this method could fail in infinitely many ways, just like `parseInt`!

From the perspective of the compiler, `fallBackTo` hasn't changed anything about

the fallibility of the `Future`. In contrast, in ZIO `parseInt` would have a type of `IO[NumberFormatException, Int]`, and `parseIntOrZero` would have a type of `UIO[Int]`, indicating precisely how `parseInt` can fail and that `parseIntOrZero` cannot fail.

2.6.3 Future Does Not Have A Way To Model The Dependencies Of An Effect

The final difference between `ZIO` and `Future` that we have seen so far is that `Future` does not have any way to model the dependencies of an effect. This requires other solutions to dependency injection, which are usually manual (they cannot be inferred) or they depend on third-party libraries.

We will spend much more time on this later in the book, but for now just note that `ZIO` has direct support for dependency injection, but `Future` does not. This means that in practice, most Future code in the real world is not very testable, because it requires too much plumbing and boilerplate.

2.7 More Effect Constructors

Earlier in this chapter, we saw how to use the `ZIO.effect` constructor to convert procedural code to ZIO effects.

The `ZIO.effect` constructor is a useful and common effect constructor, but it's not suitable for every scenario:

1. **Fallible**. The `ZIO.effect` constructor returns an effect that can fail with any kind of `Throwable` (`ZIO[Any, Throwable, A]`). This is is the right choice when you are converting legacy code into ZIO and don't know if it throws exceptions, but sometimes, we know that some code doesn't throw exceptions (like retrieving the system time).
2. **Synchronous**. The `ZIO.effect` constructor requires that our procedural code be synchronous, returning some value of the specified type from the captured block of code. But in an asynchronous API, we have to register a callback to be invoked when a value of type `A` is available. How do we convert asynchronous code to ZIO effects?
3. **Unwrapped**. The `ZIO.effect` constructor assumes the value we are computing is not wrapped in yet another data type, which has its own way of modeling failure. But some of the code that we interact with return an `Option[A]`, an `Either[E, A]`, a `Try[A]`, or even a `Future[A]`. How do we convert from these types into ZIO effects?

Fortunately, ZIO comes with robust constructors that handle custom failure scenarios, asynchronous code, and other common data types.

2.7.1 Pure Versus Impure Values

Before introducing other ZIO effect constructors, we need to first talk about *referential transparency*. An expression such as `2 + 2` is *referentially transparent* if we can always replace the computation with its result in any program while still preserving its runtime behavior.

For example, consider this expression:

```
val sum: Int = 2 + 2
```

We could replace the expression `2 + 2` with its result, `4` and the behavior of our program would not change.

In contrast, consider the following simple program that reads a line of input from the console, and then prints it out to the console:

```
import scala.io.StdIn

val echo: Unit = {
  val line = StdIn.readLine()
  println(line)
}
```

We can't replace the body of `echo` with its result and preserve the behavior of the program. The result value of `echo` is just the `Unit` value, so if we replace `echo` with its return value we would have:

```
val echo: Unit = ()
```

These two programs are definitely not the same. The first one reads input from the user and prints the input to the console, but the second program does nothing at all!

The reason we can't substitute the body of `echo` with its computed result is that it performs *side effects*. It does things *on the side* (reading from and writing to the console). This is in contrast to referentially transparent functions, which are free of side-effects, and which just compute values.

Expressions without side-effects are called *pure expressions*, while functions whose body are pure expressions are called *pure functions*.

The `ZIO.effect` constructor takes side-effecting code, and converts it into a pure value, which merely describes side-effects.

To see this in action, let's revisit the ZIO implementation for our `echo` program:

```
import zio._

val readLine =
  ZIO.effect(StdIn.readLine())
```

```
def printLine(line: String) =
  ZIO.effect(println(line))

val echo =
  for {
    line <- readLine
    _    <- printLine(line)
  } yield ()
```

This program is referentially transparent because it just builds up a blueprint (an immutable value that describes a workflow), without performing any side-effects. We can replace the code that builds up this blueprint with the resulting blueprint and we still just have a plan for this `echo` program.

So we can view referential transparency as another way of looking at the idea of functional effects as blueprints. Functional effects make side-effecting code referentially transparent by describing their side-effects, instead of performing them.

This separation between description and execution untangles the *what* from the *how*, and gives us enormous power to transform and compose effects, as we will see over the course of this book.

Referential transparency is an important concept when converting code to ZIO, because if a value or a function is referentially transparent, then we don't need to convert it into a ZIO effect. However, if it's impure, then we need to convert it into a ZIO effect by using the right effect constructor.

ZIO tries to do the right thing even if you accidentally treat side-effecting code as pure code. But mixing side-effecting code with ZIO code can be a source of bugs, so it is best to be careful about using the right effect constructor. As a side benefit, this will make your code easier to read and review for your colleagues.

2.7.2 Effect Constructors For Pure Computations

ZIO comes with a variety of effect constructors to convert pure values into ZIO effects. These constructors are useful primarily when combining other ZIO effects, which have been constructed from side-effecting code, with pure code.

In addition, even pure code can benefit from some features of ZIO, such as environment, typed errors, and stack safety.

The two most basic ways to convert pure values into ZIO effects are `succeed` and `fail`:

```
object ZIO {
  def fail[E](e: => E): ZIO[Any, E, Nothing] = ???
  def succeed[A](a: => A): ZIO[Any, Nothing, A] = ???
}
```

The `ZIO.succeed` constructor converts a value into an effect that succeeds with that value. For example, `ZIO.succced(42)` constructs an effect that succeeds with the value 42. The failure type of the effect returned by `ZIO.succeed` is `Nothing`, because effects created with this constructor cannot fail.

The `ZIO.fail` constructor converts a value into an effect that fails with that value. For example, `ZIO.fail(new Exception)` construct an effect that fails with the specified exception. The success type of the effect returned by `ZIO.fail` is `Nothing`, because effects created with this constructor cannot succeed.

We will see that effects which cannot succeed, either because they fail or because they run forever, often use `Nothing` as the success type.

In addition to these two basic constructors, there are a variety of other constructors that can convert standard Scala data types into ZIO effects.

```
import scala.util.Try

object ZIO {
  def fromEither[E, A](eea: => Either[E, A]): IO[E, A] = ???
  def fromOption[A](oa: => Option[A]): IO[None.type, A] = ???
  def fromTry[A](a: => Try[A]): Task[A] = ???
}
```

These constructors translate the success and failure cases of the original data type to the ZIO success and error types.

The `ZIO.fromEither` constructor converts an `Either[E, A]` into an `IO[E, A]` effect. If the `Either` is a `Left`, then the resulting ZIO effect will fail with an `E`, but if it is a `Right`, then the resulting ZIO effect will succeed with an `A`.

The `ZIO.fromTry` constructor is similar, except the error type is fixed to `Throwable`, because a `Try` can only fail with `Throwable`.

The `ZIO.fromOption` constructor is more interesting and illustrates an idea that will come up often. Notice that the error type is `None.type`. This is because an `Option` only has one failure mode. Either an `Option[A]` is a `Some[A]` with a value or it is a `None`, with no other information.

So an `Option` can fail, but there is essentially only one way it could ever fail—with the value `None`. The type of this lone failure value is `None.type`.

These are not the only effect constructors for pure values. In the exercises at the end of this chapter, you will explore a few of the other constructors.

2.7.3 Effect Constructors for Side Effecting Computations

The most important effect constructors are those for side-effecting computations. These constructors convert procedural code into ZIO effects, so they become blueprints that separate the *what* from the *how*.

Earlier in this chapter, we introduced `ZIO.effect`. This constructor captures side-effecting code, and defers its evaluation until later, translating any exceptions through in the code into `ZIO.fail` values.

Sometimes, however, we want to convert side-effecting code into a ZIO effect, but we know the side-effecting code does not throw any exceptions. For example, checking the system time or generating a random variable are definitely side-effects, but they cannot throw exceptions.

For these cases, we can use the constructor `ZIO.effectTotal`, which converts procedural code into a ZIO effect that cannot fail:

```
object ZIO {
  def effectTotal[A](a: => A): ZIO[Any, Nothing, A]
}
```

2.7.3.1 Converting Async Callbacks

A lot of code in the JVM ecosystem is non-blocking. Non-blocking code doesn't synchronously compute and return a value. Instead, when you call an asynchronous function, you must provide a callback, and then later, when the value is available, your callback will be invoked with the value. (Sometimes this is hidden behind `Future` or some other asynchronous data type.)

For example, let's say we have the non-blocking query API shown in the following snippet:

```
def getUserByIdAsync(id: Int)(cb: Option[String] => Unit): Unit =
  ???
```

If we give this function an `id` that we are interested in, it will look up the user in the background, but return right away. Then later, when the user has been retrieved, it will invoke the callback function that we pass to the method.

The use of `Option` in the type signature indicates that there may not be a user with the `id` we requested.

In the following code snippet, we call `getUserByIdAsync`, and pass a callback that will simply print out the name of the user when it is received:

```
getUserByIdAsync(0) {
  case Some(name) => println(name)
  case None =>       println("User not found!")
}
```

Notice that the call to `getUserByIdAsync` will return almost immediately, even though it will be some time (maybe even seconds or minutes) before our callback is invoked, and the name of the user is actually printed to the console.

Callback based APIs can improve performance, because we can write more efficient code that doesn't waste threads. But working directly with callback-

based asynchronous code can be quite painful, leading to highly nested code, making it difficult to propagate success and error information to the right place, and making it impossible to handle resources safely.

Fortunately, like Scala's `Future` before it, ZIO allows us to take asynchronous code, and convert it to ZIO functional effects.

The constructor we need to perform this conversion is `ZIO.effectAsync`, and its type signature is shown in the following snippet:

```
object ZIO {
  def effectAsync[R, E, A](
    cb: (ZIO[R, E, A] => Unit) => Any
  ): ZIO[R, E, A] =
    ???
}
```

The type signature of `ZIO.effectAsync` can be tricky to understand, so let's look at an example.

To convert the `getUserByIdAsync` procedural code into ZIO, we can use the `ZIO.effectAsync` constructor as follows:

```
def getUserById(id: Int): ZIO[Any, None.type, String] =
  ZIO.effectAsync { callback =>
    getUserByIdAsync(id) {
      case Some(name) => callback(ZIO.succeed(name))
      case None       => callback(ZIO.fail(None))
    }
  }
```

The callback provided by `effectAsync` expects a ZIO effect, so if the user exists in the database, we convert the username into a ZIO effect using `ZIO.succeed`, and then invoke the callback with this successful effect. On the other hand, if the user does not exist, we convert `None` into a ZIO effect using `ZIO.fail`, and we invoke the callback with this failed effect.

We had to work a little to convert this asynchronous code into a ZIO function, but now we never need to deal with callbacks when working with this query API. We can now treat `getUserById` like any other ZIO function, and compose its return value with methods like `flatMap`, all without ever blocking, and with all of the guarantees that ZIO provides us around resource safety.

As soon as the result of the `getUserById` computation is available, we will just continue with the other computations in the blueprint we have created.

Note here that in `effectAsync` the callback function may only be invoked once, so it's not appropriate for converting all asynchronous APIs. If the callback may be invoked more than once, you can use the `effectAsync` constructor on `ZStream`, discussed later in this book.

The final constructor we will cover in this chapter is `ZIO.fromFuture`, which converts a function that creates a `Future` into a ZIO effect.

The type signature of this constructor is as follows:

```
def fromFuture[A](make: ExecutionContext => Future[A]): Task[A] =
  ???
```

Because a `Future` is a running computation, we have to be quite careful in how we do this. The `fromFuture` constructor doesn't take a `Future`. Rather, the constructor takes a function `ExecutionContext => Future[A]`, which describes how to make a `Future` given an `ExecutionContext`.

Although you don't need to use the provided `ExecutionContext` when you convert a `Future` into a ZIO effect, if you do use the context, then ZIO can manage where the `Future` runs at higher-levels.

If possible, we want to make sure that our implementation of the `make` function creates a new `Future`, instead of returning a `Future` that is already running. The following code shows an example of doing just this:

```
def goShoppingFuture(
  implicit ec: ExecutionContext
): Future[Unit] =
  Future(println("Going to the grocery store"))

val goShopping: Task[Unit] =
  Task.fromFuture(implicit ec => goShoppingFuture)
```

There are many other constructors to create ZIO effects from other data types such as `java.util.concurrent.Future`, and third-party packages to provide conversion from Monix, Cats Effect, and other data types.

2.8 Standard ZIO Services

Earlier in this chapter, we talked about the ZIO environment type, but we haven't used it to write any programs. We will cover the environment in depth later in this book, and show how the environment provides a comprehensive solution for the dependency injection problem.

For now, we will talk about the basic *services* that ZIO provides for every application and how we can use them. *Services* provide well-defined interfaces that can be implemented differently in testing environments and production environments.

ZIO provides four to five different default services for all applications, depending on the platform:

1. **Clock**. Provides functionality related to time and scheduling. If you are accessing the current time or scheduling a computation to occur at some

point in the future you are using this.
2. **Console**. Provides functionality related to console input and output.
3. **System**. Provides functionality for getting system and environment variables.
4. **Random**. Provides functionality for generating random values.
5. **Blocking**. Provides functionality for running blocking tasks on a separate `Executor` optimized for these kinds of workloads. Because blocking is not supported on Scala.js, this service is only available on the JVM.

Because this essential system functionality is provided as services, you can trivially test any code that uses these services, without actually interacting with production implementations.

For example, the `Random` services allows us to generate random numbers. The `Live` implementation of that service just delegates to `scala.util.Random`. But that may not always be the implementation we want. `scala.util.Random` is non-deterministic, which can be useful in production but can make it harder for us to test our programs.

For testing the random service, we may want to use a purely functional pseudo-random number generator, which always generates the same values given the same initial seed. This way, if a test fails, we can reproduce the failure and debug it.

ZIO Test, a toolkit for testing ZIO applications that we will discuss in a later chapter, provides a `TestRandom` that does exactly this. In fact, ZIO Test provides a test implementation of each of the standard services, and you can imagine wanting to provide other implementations, as well.

ZIO provides a default implementation of an `Executor` optimized for blocking tasks, but there are a variety of other ways you could tune an `Executor` for specific usage patterns.

By defining functionality in terms of well-defined interfaces, we defer concrete implementations until later. As you will see, using these services with ZIO is very easy, but at the same time, power users have tremendous flexibility to provide custom implementations of these services (or those you define in your own application).

The second, smaller benefit of using services is that you can use the feature to document which capabilities are being used in an effect. If we see an effect with a type signature of `ZIO[Clock, Nothing, Unit]` we can conclude that this effect, which cannot fail, is using functionality related to time or scheduling (probably not functionality related to random numbers).

However, this benefit is smaller because it is true only insofar as your team is very disciplined about coding to interfaces, instead of implementations. Because effect constructors can wrap any side-effecting code into a ZIO value, there is nothing stopping us from writing code like the following:

```
val int: ZIO[Any, Nothing, Int] =
  ZIO.effectTotal(scala.util.Random.nextInt())
```

In this snippet, we are generating random numbers, but this is not reflected in the type signature because we are wrapping `scala.util.Random` directly, instead of using the methods on the `Random` service. Unfortunately, the compiler cannot check this for us, so ensuring developers code to interfaces must be enforced with code review.

As such, we consider the ability to "see" what capabilities a computation uses as a *secondary* and optional benefit of using services. The primary benefit is just being able to plug in different implementations in testing and production environments.

Now let's discuss each of the standard ZIO services in some detail.

2.8.1 Clock

The `Clock` service provides functionality related to time and scheduling. This includes several methods to obtain the current time in different ways (`currentTime` to return the current time in the specified `TimeUnit`, `currentDateTime` to return the current `OffsetDateTime`, and `nanoTime` to obtain the current time in nanoseconds).

In addition, the `Clock` service includes a `sleep` method, which can be used to sleep for a certain amount of time.

The signature of `nanoTime` and `sleep` are shown in the following snippet:

```
import zio.duration._

package object clock {
  def nanoTime: URIO[Clock, Long]
  def sleep(duration: => Duration): URIO[Clock, Unit]
}
```

The `sleep` method is particularly important. It does not complete execution until the specified duration has elapsed, and like all ZIO operations, it is non-blocking, so it doesn't actually consume any threads while it is waiting for the time to elapse.

We could use the `sleep` method to implement the `delay` operator that we saw earlier in this chapter:

```
import zio.clock._
import zio.duration._

def delay[R, E, A](zio: ZIO[R, E, A])(
  duration: Duration
```

```
): ZIO[R with Clock, E, A] =
  clock.sleep(duration) *> zio
```

The `Clock` service is the building block for all time and scheduling functionality in ZIO. Consequently, you will see the `Clock` service as a component of the environment whenever working with retrying, repetition, timing, or other features related to time and scheduling built into ZIO.

2.8.2 Console

The `Console` service provides functionality around reading from and writing to the console.

So far in this book, we have been interacting with the console by converting procedural code in the Scala library to ZIO effects, using the `ZIO.effect` constructor. This was useful to illustrate how to translate procedural to ZIO, and demonstrate there is no "magic" in ZIO's own console facilities.

However, wrapping console functionality directly is not ideal, because we cannot provide alternative implementations for testing environments. In addition, there are some tricky edge corner cases for console interaction that the `Console` services handles for us. (For example, reading from the console can fail only with an `IOException`.)

The key methods on the `Console` service are `getStrLn`, which is analogous to `readLine()` and `putStrLn`, which is the equivalent of `println`. There is also a `putStr` method if you do not want to add a newline after printing text to the console.

```
package object console {
  val getStrLn: ZIO[Console, IOException, String]
  def putStr(line: => String): URIO[Console, Unit]
  def putStrLn(line: => String): URIO[Console, Unit]
```

The `Console` service is commonly used in console applications, but is less common in generic code than `Clock` or `Random`.

In the rest of this book, we will illustrate examples involving console applications with these methods, rather than converting methods from the Scala standard library.

2.8.3 System

The `System` service provides functionality to get system and environment variables.

```
package object system {
  def env(variable: String): IO[SecurityException, Option[String]]
  def property(prop: String): IO[Throwable, Option[String]]
```

The two main methods on the `System` service are `env`, which accesses a specified environment variable, and `property`, which accesses a specified system property. There are also other variants for obtaining all environment variables or system properties, or specifying a backup value, if a specified environment variable or property does not exist.

Like the `Console` service, the `System` service tends to be used more in applications or certain libraries (e.g. those dealing with configuration) but is uncommon in generic code.

2.8.4 Random

The `Random` service provides functionality related to random number generation. The `Random` service exposes essentially the same interface as `scala.util.Random`, but all the methods return functional effects. So if you're familiar with code like `random.nextInt(6)` from the standard library, you should be very comfortable working with the `Random` service.

The `Random` service is sometimes used in generic code in scheduling, such as when adding a random delay between recurrences of some effect.

2.8.5 Blocking

The `Blocking` service supports running blocking effects on an `Executor` optimized for blocking tasks. By default, the ZIO runtime is optimized for asynchronous and computationally-bound tasks, with a small fixed number of threads that perform all work.

Although this choice optimizes throughput for async and CPU tasks, it means that you could potentially exhaust all of ZIO's default threads if you run blocking I/O operations on them. Therefore, it is critical that blocking tasks be run on a separate blocking thread pool, which is optimized for these workloads.

The `Blocking` service has several methods to support this use case.

The most fundamental is the `blocking` operator, which takes an effect and ensures it will be run on the blocking thread pool.

```
import zio.blocking._

object blocking {
  def blocking[R <: Blocking, E, A](
    zio: ZIO[R, E, A]
  ): ZIO[R, E, A] =
    ???
}
```

Let's say that the database query described above was not implemented in terms of a callback based API but instead synchronously waited until the result was

available (thereby blocking the calling thread). In this case, we would want to make sure the effect is executed on the blocking thread pool by using the `blocking` operator, as shown in the following snippet:

```
import zio.blocking._

def getUserById(id: Int): IO[Unit, String] =
  ???

def getUserByIdBlocking(id: Int): ZIO[Blocking, Unit, String] =
  blocking(getUserById(id))
```

Notice that the `Blocking` service is now a dependency in the type signature, clarifying that the returned effect involves blocking IO, in a way its previous signature did not.

If you want to construct an effect from a blocking side-effect directly, you can use the `effectBlocking` constructor, which is equivalent to `blocking(ZIO.effect(...))`.

2.9 Recursion And ZIO

We talked earlier in this chapter about using `flatMap` and related operators to compose effects sequentially.

Ordinarily, if you call a recursive function, and it recurses deeply, the thread running the computation may run out of stack space, which will result in your program throwing a stack overflow exception.

One of the features of ZIO is that ZIO effects are stack-safe for arbitrarily recursive effects. So we can write ZIO functions that call themselves to implement any kind of recursive logic with ZIO.

For example, let's say we want to implement a simple console program that gets two integers from the user and multiplies them together.

We can start by implementing an operator to get a single integer from the user, as shown in the following snippet:

```
import zio.console._

val readInt: RIO[Console, Int] =
  for {
    line <- console.getStrLn
    int  <- ZIO.effect(line.toInt)
  } yield int
```

This effect can fail with an error type of `Throwable`, because the input from the user might not be a valid integer. If the user doesn't enter a valid integer, we

want to print a helpful error message to the user, and then try again.

We can build this functionality atop our existing `readInt` effect by using recursion. We define a new effect `readIntOrRetry` that will first call `readInt`. If `readInt` is successful, we just return the result. If not, we prompt the user to enter a valid integer, and then recurse:

```
lazy val readIntOrRetry: URIO[Console, Int] =
  readInt
    .orElse(console.putStrLn("Please enter a valid integer")
    .zipRight(readIntOrRetry))
```

Using recursion, we can create our own sophisticated control flow constructs for our ZIO programs.

2.10 Conclusion

Functional effects are blueprints for concurrent workflows, immutable values that offer a variety of operators for transforming and combining effects to solve more complex problems.

The ZIO type parameters allow us to model effects that require context from an environment before they can be executed; they allow us to model failure modes (or a lack of failure modes); and they allow us to describe the final successful result that will be computed by an effect.

ZIO offers a variety of ways to create functional effects from synchronous code, asynchronous code, pure computations, and impure computations. In addition, ZIO effects can be created from other data types built into the Scala standard library.

ZIO uses the environment type parameter to make it easy to write testable code that interacts with interfaces, without the need to manually propagate those interfaces throughout the entire application. Using this type parameter, ZIO ships with standard services for interacting with the console, the system, random number generation, and a blocking thread pool.

With these tools, you should be able to write your own simple ZIO programs, convert existing code you have written into `ZIO` using effect constructors, and leverage the functionality built into ZIO.

2.11 Exercises

1. Implement a ZIO version of the function `readFile` by using the `ZIO.effect` constructor.

```
def readFile(file: String): String = {
  val source = scala.io.Source.fromFile(file)
```

```
  try source.getLines.mkString finally source.close()
}

def readFileZio(file: String) = ???
```

2. Implement a ZIO version of the function `readFile` by using the `ZIO.effect` constructor.

```
def writeFile(file: String, text: String): Unit = {
  import java.io._
  val pw = new PrintWriter(new File(file))
  try pw.write(text) finally pw.close
}

def writeFileZio(file: String, text: String) = ???
```

3. Using the `flatMap` method of ZIO effects, together with the `readFileZio` and `writeFileZio` functions that you wrote, implement a ZIO version of the function `copyFile`.

```
def copyFile(source: String, dest: String): Unit = {
  val contents = readFile(source)
  writeFile(dest, contents)
}

def copyFileZio(source: String, dest: String) = ???
```

4. Rewrite the following ZIO code that uses `flatMap` into a *for comprehension*.

```
def printLine(line: String) = ZIO.effect(println(line))
val readLine = ZIO.effect(scala.io.StdIn.readLine())

printLine("What is your name?").flatMap(_ =>
  readLine.flatMap(name =>
    printLine(s"Hello, ${name}!")))
```

5. Rewrite the following ZIO code that uses `flatMap` into a *for comprehension*.

```
val random = ZIO.effect(scala.util.Random.nextInt(3) + 1)
def printLine(line: String) = ZIO.effect(println(line))
val readLine = ZIO.effect(scala.io.StdIn.readLine())

random.flatMap(int =>
  printLine("Guess a number from 1 to 3:").flatMap(_ =>
    readLine.flatMap(num =>
      if (num == int.toString) printLine("You guessed right!")
      else printLine(s"You guessed wrong, the number was ${int}!")))
)
```

6. Implement the `zipWith` function in terms of the toy model of a ZIO effect. The function should return an effect that sequentially composes the specified effects, merging their results with the specified user-defined function.

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A])

def zipWith[R, E, A, B, C](
  self: ZIO[R, E, A],
  that: ZIO[R, E, B]
)(f: (A, B) => C): ZIO[R, E, C] =
    ???
```

7. Implement the `collectAll` function in terms of the toy model of a ZIO effect. The function should return an effect that sequentially collects the results of the specified collection of effects.

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A])

def collectAll[R, E, A](
  in: Iterable[ZIO[R, E, A]]
): ZIO[R, E, List[A]] =
  ???
```

8. Implement the `foreach` function in terms of the toy model of a ZIO effect. The function should return an effect that sequentially runs the specified function on every element of the

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A])

def foreach[R, E, A, B](
  in: Iterable[A]
)(f: A => ZIO[R, E, B]): ZIO[R, E, List[B]] =
    ???
```

9. Implement the `orElse` function in terms of the toy model of a ZIO effect. The function should return an effect that tries the left hand side, but if that effect fails, it will fallback to the effect on the right hand side.

```
final case class ZIO[-R, +E, +A](run: R => Either[E, A])

def orElse[R, E1, E2, A](
  self: ZIO[R, E1, A],
  that: ZIO[R, E2, A]
): ZIO[R, E2, A] =
  ???
```

10. Using the following code as a foundation, write a ZIO application that prints out the contents of whatever files are passed into the program as command-line arguments. You should use the functions `readFileZio`

and `writeFileZio` that you developed in these exercises, as well as `ZIO.foreach`.

```
import zio.{ App => ZIOApp }

object Cat extends ZIOApp {
  def run(commandLineArguments: List[String]) = ???
}
```

11. Using `ZIO.fail` and `ZIO.succeed`, implement the following function, which converts an `Either` into a ZIO effect:

```
def eitherToZIO[E, A](either: Either[E, A]): ZIO[Any, E, A] = ???
```

12. Using `ZIO.fail` and `ZIO.succeed`, implement the following function, which converts a `List` into a ZIO effect, by looking at the head element in the list and ignoring the rest of the elements.

```
def listToZIO[A](list: List[A]): ZIO[Any, None.type, A] = ???
```

13. Using `ZIO.effectTotal`, convert the following procedural function into a ZIO function:

```
def currentTime(): Long = System.currentTimeMillis()

lazy val currentTimeZIO: ZIO[Any, Nothing, Long] = ???
```

14. Using `ZIO.effectAsync`, convert the following asynchronous, callback-based function into a ZIO function:

```
def getCacheValue(
  key: String,
  onSuccess: String => Unit,
  onFailure: Throwable => Unit
): Unit =
  ???

def getCacheValueZio(key: String): ZIO[Any, Throwable, String] = ???
```

15. Using `ZIO.effectAsync`, convert the following asynchronous, callback-based function into a ZIO function:

```
trait User

def saveUserRecord(
  user: User,
  onSuccess: () => Unit,
  onFailure: Throwable => Unit
): Unit =
  ???
```

```
def saveUserRecordZio(user: User): ZIO[Any, Throwable, Unit] = ???
```

16. Using `ZIO.fromFuture`, convert the following code to ZIO:

```
import scala.concurrent.{ ExecutionContext, Future }
trait Query
trait Result

def doQuery(query: Query)(
  implicit ec: ExecutionContext): Future[Result] =
    ???

def doQueryZio(query: Query): ZIO[Any, Throwable, Result] =
  ???
```

17. Using the `Console`, write a little program that asks the user what their name is, and then prints it out to them with a greeting.

```
import zio. { App => ZIOApp }

object HelloHuman extends ZIOApp {
  def run(args: List[String]) = ???
}
```

18. Using the `Console` and `Random` services in ZIO, write a little program that asks the user to guess a randomly chosen number between 1 and 3, and prints out if they were correct or not.

```
import zio._

object NumberGuessing extends ZIOApp {
  def run(args: List[String]) = ???
}
```

19. Using the `Console` service and recursion, write a function that will repeatedly read input from the console until the specified user-defined function evaluates to `true` on the input.

```
import java.io.IOException

import zio.console._

def readUntil(
  acceptInput: String => Boolean
): ZIO[Console, IOException, String] =
  ???
```

20. Using recursion, write a function that will continue evaluating the specified effect, until the specified user-defined function evaluates to `true` on the

output of the effect.

```
def doWhile[R, E, A](
  body: ZIO[R, E, A]
)(condition: A => Boolean): ZIO[R, E, A] =
  ???
```